9
Adding Real-Time Bidding Capabilities to the Marketplace

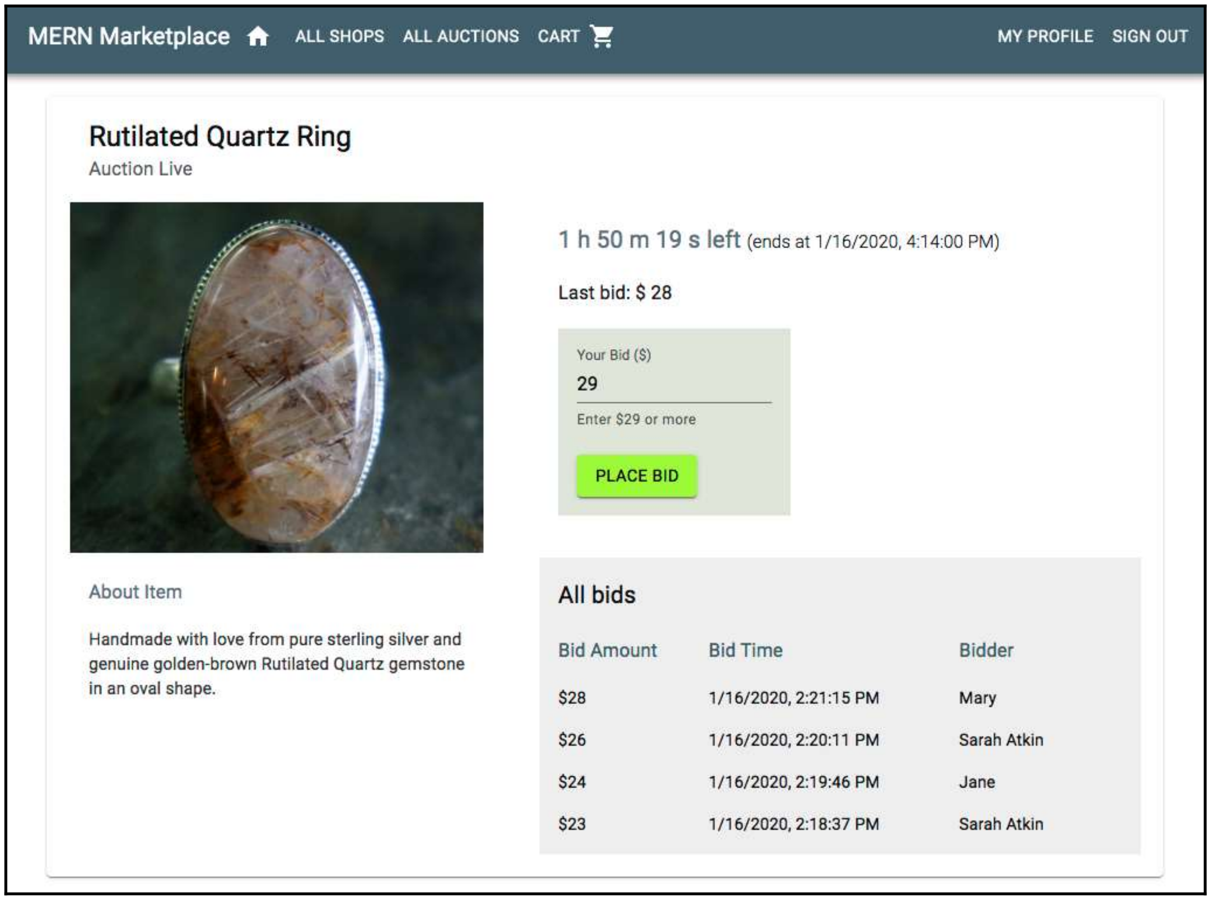
In a world more connected than ever before, instant communication and real-time updates are expected behaviors in any application that enables interaction between users. Adding real-time features to your application can keep your users engaged, and because of that, they will be spending more time on your platform. In this chapter, we will learn how to use the MERN stack technologies, along with Socket.IO, to easily integrate real-time behavior in a full-stack application. We will do this by incorporating an auctioning feature with real-time bidding capabilities in the MERN Marketplace application that we developed in `Chapter 7`, *Exercising MERN Skills with an Online Marketplace*, and `Chapter 8`, *Extending the Marketplace for Orders and Payments*. After going through the implementation of this auction and bidding feature, you will know how to utilize sockets in a MERN stack application to add real-time features of your choice.

In this chapter, we will extend the online marketplace application by covering the following topics:

- Introducing real-time bidding in the MERN Marketplace
- Adding auctions to the marketplace
- Displaying the auction view
- Implementing real-time bidding with Socket.IO

Introducing real-time bidding in the MERN Marketplace

The MERN Marketplace application already allows its users to become sellers and maintain shops with products that can be bought by regular users. In this chapter, we will extend these functionalities to allow sellers to create auctions for items that other users can place bids on in a fixed duration of time. The auction view will describe the item for sale and let signed in users place bids when the auction is live. Different users can place their own bids, and also see other users placing bids in real-time, with the view updating accordingly. The completed auction view, with an auction in a live state, will render as follows:



Rutilated Quartz Ring
Auction Live

1 h 50 m 19 s left (ends at 1/16/2020, 4:14:00 PM)

Last bid: $ 28

Your Bid ($)
29
Enter $29 or more

PLACE BID

About Item

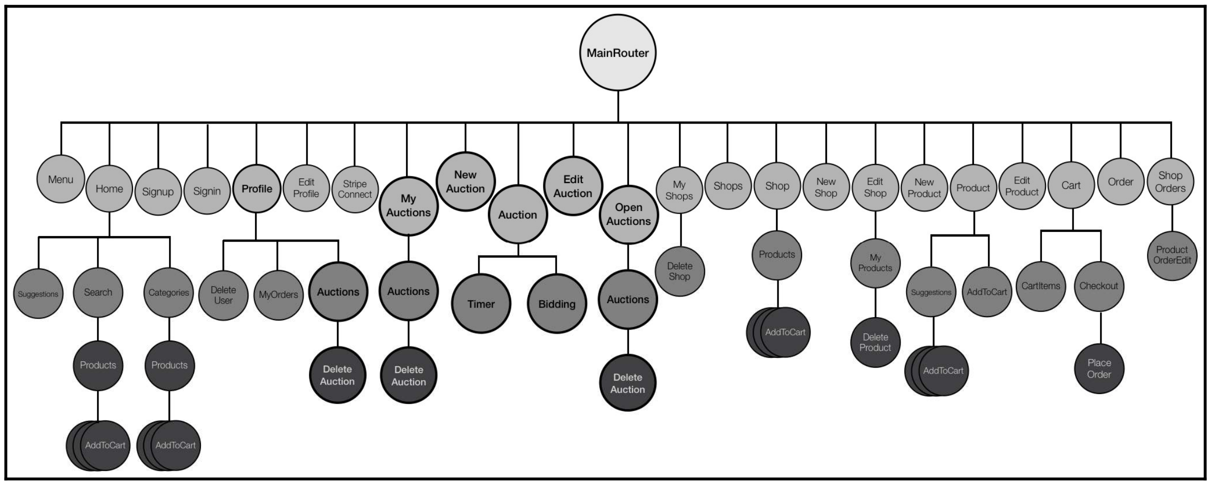
Handmade with love from pure sterling silver and genuine golden-brown Rutilated Quartz gemstone in an oval shape.

All bids

Bid Amount	Bid Time	Bidder
$28	1/16/2020, 2:21:15 PM	Mary
$26	1/16/2020, 2:20:11 PM	Sarah Atkin
$24	1/16/2020, 2:19:46 PM	Jane
$23	1/16/2020, 2:18:37 PM	Sarah Atkin

The code for the complete MERN Marketplace application is available on GitHub at `https://github.com/PacktPublishing/Full-Stack-React-Projects-Second-Edition/tree/master/Chapter09/mern-marketplace-bidding`. The implementations discussed in this chapter can be accessed in the *bidding* branch of the repository. You can clone this code and run the application as you go through the code explanations in the rest of this chapter.

The following component tree diagram shows the custom components that make up the entire MERN Marketplace frontend, including components for the auction and bidding-related features that will be implemented in the rest of this chapter:



The features that will be discussed in this chapter modify some of the existing components, such as `Profile` and `Menu`, and also add new components, such as `NewAuction`, `MyAuctions`, `Auction`, and `Bidding`. In the next section, we will begin extending this online marketplace by integrating the option to add auctions to the platform.

Adding auctions to the marketplace

In the MERN Marketplace, we will allow a user who is signed in and has an active seller account to create auctions for items that they want other users to place bids on. To enable the features of adding and managing auctions, we will need to define how to store auction details and implement the full-stack slices that will let users create, access and update auctions on the platform. In the following sections, we will build out this auction module for the application. First, we will define the auction model with a Mongoose Schema for storing details about each auction. Then, we will discuss implementations for the backend APIs and frontend views that are needed to create new auctions, list auctions that are ongoing, created by the same seller and bid on by the same user, and modify existing auctions by either editing details of, or deleting an auction from the application.

Defining an Auction model

We will implement a Mongoose model that will define an Auction model for storing the details of each auction. This model will be defined in `server/models/auction.model.js`, and the implementation will be similar to other Mongoose model implementations we've covered in previous chapters, such as the Shop model we defined in `Chapter 7`, *Exercising MERN Skills with an Online Marketplace*. The Auction Schema in this model will have fields to store auction details such as the name and description of the item being auctioned, along with an image and a reference to the seller creating this auction. It will also have fields that specify the start and end time for bidding on this auction, a starting value for bids, and the list of bids that have been placed for this auction. The code for defining these auction fields is as follows:

- **Item name and description**: The auction item name and description fields will be string types, with `itemName` as a required field:

```
itemName: {
    type: String,
    trim: true,
    required: 'Item name is required'
},
description: {
    type: String,
    trim: true
},
```

- **Item image**: The `image` field will store the image file representing the auction item so that it can be uploaded by the user and stored as data in the MongoDB database:

```
image: {
    data: Buffer,
    contentType: String
},
```

- **Seller**: The `seller` field will reference the user who is creating the auction:

```
seller: {
    type: mongoose.Schema.ObjectId,
    ref: 'User'
},
```

- **Created and updated at times**: The `created` and `updated` fields will be `Date` types, with `created` generated when a new auction is added, and `updated` changed when any auction details are modified:

```
updated: Date,
created: {
    type: Date,
    default: Date.now
},
```

- **Bidding start time**: The `bidStart` field will be a `Date` type that will specify when the auction goes live so that users can start placing bids:

```
bidStart: {
    type: Date,
    default: Date.now
},
```

- **Bidding end time**: The `bidEnd` field will be a `Date` type that will specify when the auction ends, after which the users cannot place bids on this auction:

```
bidEnd: {
   type: Date,
   required: "Auction end time is required"
},
```

- **Starting bid**: The `startingBid` field will store values of the `Number` type, and it will specify the starting price for this auction:

  ```
  startingBid: {
      type: Number,
      default: 0
  },
  ```

- **List of bids**: The `bids` field will be an array containing details of each bid placed against the auction. When we store bids in this array, we will push the latest bid to the beginning of the array. Each bid will contain a reference to the user placing the bid, the bid amount the user offered, and the timestamp when the bid was placed:

  ```
  bids: [{
      bidder: {type: mongoose.Schema.ObjectId, ref: 'User'},
      bid: Number,
      time: Date
  }]
  ```

These auction-related fields will allow us to implement auction and bidding-related features for the MERN Marketplace application. In the next section, we will start developing these features by implementing the full-stack slice, which will allow sellers to create new auctions.

Creating a new auction

For a seller to be able to create a new auction on the platform, we will need to integrate a full-stack slice that allows the user to fill out a form view in the frontend, and then save the entered details to a new auction document in the database in the backend. To implement this feature, in the following sections, we will add a create auction API in the backend, along with a way to fetch this API in the frontend, and a create new auction form view that takes user input for auction fields.

The create auction API

For the implementation of the backend API, which will allow us to create a new auction in the database, we will declare a POST route, as shown in the following code.

`mern-marketplace/server/routes/auction.routes.js`:

```
router.route('/api/auctions/by/:userId')
  .post(authCtrl.requireSignin, authCtrl.hasAuthorization,
       userCtrl.isSeller, auctionCtrl.create)
```

A POST request to this route at `/api/auctions/by/:userId` will ensure the requesting user is signed in and is also authorized. In other words, it is the same user associated with the `:userId` specified in the route param. Then, before creating the auction, it is checked if this given user is a seller using the `isSeller` method that's defined in the user controller methods.

To process the `:userId` parameter and retrieve the associated user from the database, we will utilize the `userByID` method from the user controller methods. We will add the following to the `Auction` routes in `auction.routes.js` so that the user is available in the `request` object as `profile`.

`mern-marketplace/server/routes/auction.routes.js`:

```
router.param('userId', userCtrl.userByID)
```

The `auction.routes.js` file, which contains the auction routes, will be very similar to the `user.routes` file. To load these new auction routes in the Express app, we need to mount the auction routes in `express.js`, as we did for the auth and user routes.

`mern-marketplace/server/express.js`:

```
app.use('/', auctionRoutes)
```

The `create` method in the auction controller, which is invoked after a seller is verified, uses the `formidable` node module to parse the multipart request that may contain an image file uploaded by the user for the item image. If there is a file, `formidable` will store it temporarily in the filesystem, and we will read it using the `fs` module to retrieve the file type and data so that we can store it in the `image` field in the auction document.

The `create` controller method will look as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

```
const create = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let auction = new Auction(fields)
    auction.seller= req.profile
    if(files.image){
      auction.image.data = fs.readFileSync(files.image.path)
      auction.image.contentType = files.image.type
    }
    try {
      let result = await auction.save()
      res.status(200).json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}
```

The item image file for the auction is uploaded by the user and stored in MongoDB as data. Then, in order to be shown in the views, it is retrieved from the database as an image file at a separate GET API. The GET API is set up as an Express route at `/api/auctions/image/:auctionId`, which gets the image data from MongoDB and sends it as a file in the response. The implementation steps for file upload, storage, and retrieval are outlined in detail in the *Upload profile photo* section in `Chapter 5`, *Growing the Skeleton into a Social Media Application*.

This create auction API endpoint can now be used in the frontend to make a POST request. Next, we will add a fetch method on the client-side to make this request from the application's client interface.

Fetching the create API in the view

In the frontend, to make a request to this create API, we will set up a `fetch` method on the client-side to make a POST request to the API route and pass it the multipart form data containing details of the new auction in the `body`. This fetch method will be defined as follows.

`mern-marketplace/client/auction/api-auction.js`:

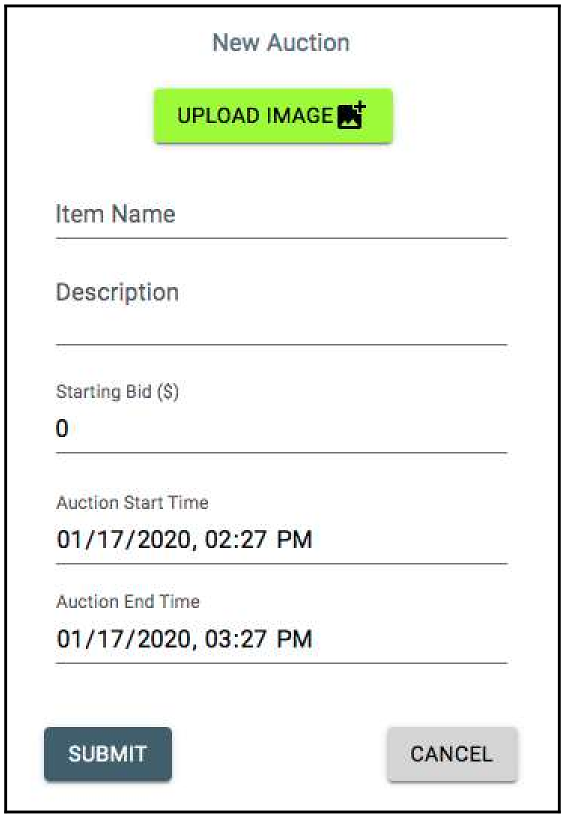
```
const create = (params, credentials, auction) => {
  return fetch('/api/auctions/by/'+ params.userId, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Authorization': 'Bearer ' + credentials.t
      },
      body: auction
    })
    .then((response) => {
      return response.json()
    }).catch((err) => console.log(err))
}
```

The response that's received from the server will be returned to the component calling this fetch method. We will use this method in the new auction form view to send the user-entered auction details to the backend and create a new auction in the database. In the next section, we will implement this new auction form view in a React component.

The NewAuction component

Sellers in the marketplace application will interact with a form view to enter details of a new auction and create the new auction. We will render this form in the `NewAuction` component, which will allow a seller to create an auction by entering an item name and description, uploading an image file from their local filesystem, specifying the starting bid value, and creating date-time values for starting and ending bidding on this auction.

This form view will render as follows:



The implementation for this `NewAuction` component is similar to other create form implementations that we have discussed previously, such as the `NewShop` component implementation from `Chapter 7`, *Exercising MERN Skills with an Online Marketplace*. The fields that are different in this form component are the date-time input options for the auction start and end timings. To add these fields, we'll use Material-UI `TextField` components with `type` set to `datetime-local`, as shown in the following code.

`mern-marketplace/client/auction/NewAuction.js`:

```
<TextField
   label="Auction Start Time"
   type="datetime-local"
   defaultValue={defaultStartTime}
   onChange={handleChange('bidStart')}
/>
<TextField
   label="Auction End Time"
```

```
      type="datetime-local"
      defaultValue={defaultEndTime}
      onChange={handleChange('bidEnd')}
   />
```

We also assign default date-time values for these fields in the format expected by this input component. We set the default start time to the current date-time and the default end time to an hour after the current date-time, as shown here.

`mern-marketplace/client/auction/NewAuction.js`:

```
const currentDate = new Date()
const defaultStartTime = getDateString(currentDate)
const defaultEndTime = getDateString(new
Date(currentDate.setHours(currentDate.getHours()+1)))
```

The `TextField` with the type as `datetime-local` takes dates in the format `yyyy-mm-ddThh:mm`. So, we define a `getDateString` method that takes a JavaScript date object and formats it accordingly. The `getDateString` method is implemented as follows.

`mern-marketplace/client/auction/NewAuction.js`:

```
const getDateString = (date) => {
  let year = date.getFullYear()
  let day = date.getDate().toString().length === 1 ? '0' +
date.getDate() : date.getDate()
  let month = date.getMonth().toString().length === 1 ? '0' +
(date.getMonth()+1) : date.getMonth() + 1
  let hours = date.getHours().toString().length === 1 ? '0' +
date.getHours() : date.getHours()
  let minutes = date.getMinutes().toString().length === 1 ? '0' +
date.getMinutes() : date.getMinutes()
  let dateString = `${year}-${month}-${day}T${hours}:${minutes}`
  return dateString
}
```

In order to ensure the user has entered the dates correctly, with the start time set to a value before the end time, we need to add a check before submitting the form details to the backend. The validation of the date combination can be confirmed with the following code.

`mern-marketplace/client/auction/NewAuction.js`:

```
if(values.bidEnd < values.bidStart){
   setValues({...values, error: "Auction cannot end before it
starts"})
}
```

If the date combination is found to be invalid, then the user will be informed and form data will not be sent to the backend.

This `NewAuction` component can only be viewed by a signed-in user who is also a seller. Therefore, we will add a `PrivateRoute` in the `MainRouter` component. This will render this form for authenticated users at `/auction/new`.

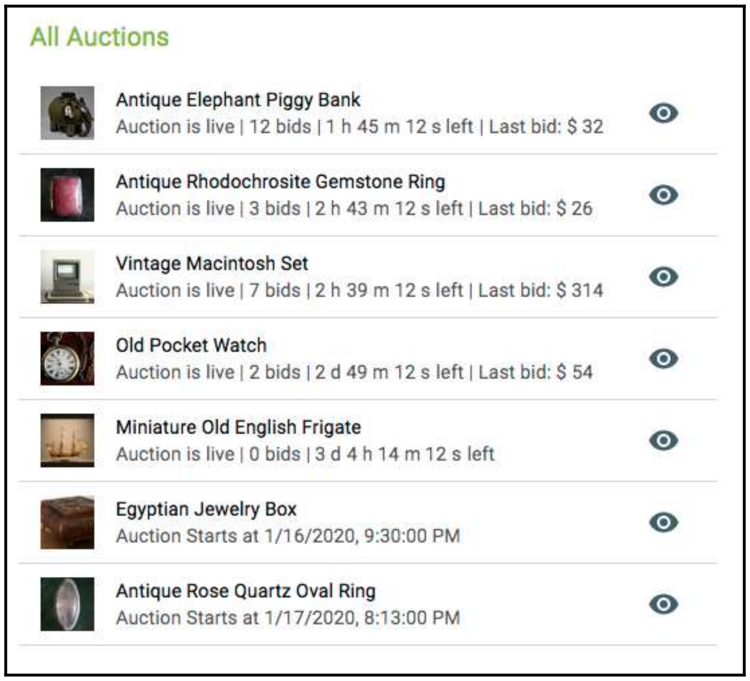
`mern-marketplace/client/MainRouter.js`:

```
<PrivateRoute path="/auction/new" component={NewAuction}/>
```

This link can be added to any of the view components that may be accessed by the seller, for example, in a view where a seller manages their auctions in the marketplace. Now that it is possible to add new auctions in the marketplace, in the next section, we will discuss how to fetch these auctions from the database in the backend so that they can be listed in the views in the frontend.

Listing auctions

In the MERN Marketplace application, we will display three different lists of auctions to the users. All users browsing through the platform will be able to view the currently open auctions, in other words, auctions that are live or are going to start at a future date. The sellers will be able to view a list of auctions that they created, while signed in users will be able to view the list of auctions they placed bids in. The list displaying the open auctions to all the users will render as follows, providing a summary of each auction, along with an option so that the user can view further details in a separate view:


All Auctions

Antique Elephant Piggy Bank
Auction is live | 12 bids | 1 h 45 m 12 s left | Last bid: $ 32

Antique Rhodochrosite Gemstone Ring
Auction is live | 3 bids | 2 h 43 m 12 s left | Last bid: $ 26

Vintage Macintosh Set
Auction is live | 7 bids | 2 h 39 m 12 s left | Last bid: $ 314

Old Pocket Watch
Auction is live | 2 bids | 2 d 49 m 12 s left | Last bid: $ 54

Miniature Old English Frigate
Auction is live | 0 bids | 3 d 4 h 14 m 12 s left

Egyptian Jewelry Box
Auction Starts at 1/16/2020, 9:30:00 PM

Antique Rose Quartz Oval Ring
Auction Starts at 1/17/2020, 8:13:00 PM

In the following sections, in order to implement these different auction lists so that they're displayed in the application, we will define the three separate backend APIs to retrieve open auctions, auctions by a seller, and auctions by a bidder, respectively. Then, we will implement a reusable React component that will take any list of auctions provided to it as a prop and render it to the view. This will allow us to display all three lists of auctions while utilizing the same component.

The open Auctions API

To retrieve the list of open auctions from the database, we will define a backend API that accepts a GET request and queries the Auction collection to return the open auctions that are found in the response. To implement this open auctions API, we will declare a route, as shown here.

`mern-marketplace/server/routes/auction.routes.js:`

```
router.route('/api/auctions')
  .get(auctionCtrl.listOpen)
```

A GET request that's received at the `/api/auctions` route will invoke the `listOpen` controller method, which will query the Auction collection in the database so that it returns all the auctions with ending dates greater than the current date.
The `listOpen` method is defined as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

```
const listOpen = async (req, res) => {
  try {
    let auctions = await Auction.find({ bidEnd: { $gt: new Date() }})
                                .sort('bidStart')
                                .populate('seller', '_id name')
                                .populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

The auctions that are returned by the query in this `listOpen` method will be sorted by the starting date, with auctions that start earlier shown first. These auctions will also contain the ID and name details of the seller and each bidder. The resulting array of auctions will be sent back in the response to the requesting client.

To fetch this API in the frontend, we will add a corresponding `listOpen` method in `api-auction.js`, similar to other API implementations. This fetch method will be used in the frontend component that displays the open auctions to the user. Next, we will implement another API to list all the auctions that a specific user placed bids in.

The Auctions by bidder API

To be able to display all the auctions that a given user placed bids in, we will define a backend API that accepts a GET request and queries the Auction collection so that it returns the relevant auctions in the response. To implement this auctions by bidder API, we will declare a route, as shown here.

`mern-marketplace/server/routes/auction.routes.js`

```
router.route('/api/auctions/bid/:userId')
  .get(auctionCtrl.listByBidder)
```

A GET request, when received at the `/api/auctions/bid/:userId` route, will invoke the `listByBidder` controller method, which will query the Auction collection in the database so that it returns all the auctions that contain bids with a bidder matching the user specified by the `userId` parameter in the route.
The `listByBidder` method is defined as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

```
const listByBidder = async (req, res) => {
  try {
    let auctions = await Auction.find({'bids.bidder':
req.profile._id})
                                .populate('seller', '_id name')
                                .populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This method will return the resulting auctions in response to the requesting client, and each auction will also contain the ID and name details of the seller and each bidder. To fetch this API in the frontend, we will add a corresponding `listByBidder` method in `api-auction.js`, similar to other API implementations. This fetch method will be used in the frontend component that displays the auctions related to a specific bidder. Next, we will implement an API that will list all the auctions that a specific seller created in the marketplace.

The Auctions by seller API

Sellers in the marketplace will see a list of auctions that they created. To retrieve these auctions from the database, we will define a backend API that accepts a GET request and queries the Auction collection so that it returns the auctions by a specific seller. To implement this auctions by seller API, we will declare a route, as shown here.

`mern-marketplace/server/routes/auction.routes.js`:

```
router.route('/api/auctions/by/:userId')
  .get(authCtrl.requireSignin, authCtrl.hasAuthorization,
       auctionCtrl.listBySeller)
```

A GET request, when received at the `/api/auctions/by/:userId` route, will invoke the `listBySeller` controller method, which will query the Auction collection in the database so that it returns all the auctions with sellers matching the user specified by the `userId` parameter in the route. The `listBySeller` method is defined as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

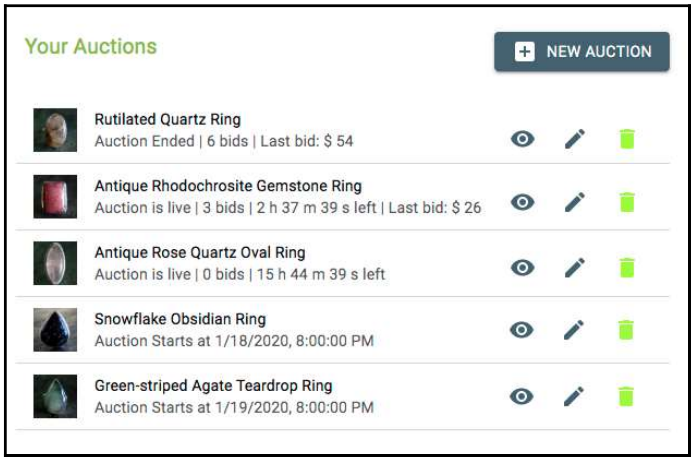
```
const listBySeller = async (req, res) => {
  try {
    let auctions = await Auction.find({seller: req.profile._id})
                                .populate('seller', '_id name')
                                .populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
```

This method will return the auctions for the specified seller in response to the requesting client, and each auction will also contain the ID and name details of the seller and each bidder.

To fetch this API in the frontend, we will add a corresponding `listBySeller` method in `api-auction.js`, similar to other API implementations. This fetch method will be used in the frontend component that displays the auctions related to a specific seller. In the next section, we will look at the implementation of the Auctions component, which will take any of these lists of auctions and display it to the end user.

The Auctions component

The different auction lists in the application will be rendered using a React component that takes an array of auction objects as props. We will implement this reusable `Auctions` component and add it to the views that will retrieve and display either the open auctions, auctions by a bidder, or auctions by a seller. The view that retrieves and renders the list of auctions created by a specific seller using the `Auctions` component will look as follows:



Rutilated Quartz Ring
Auction Ended | 6 bids | Last bid: $ 54

Antique Rhodochrosite Gemstone Ring
Auction is live | 3 bids | 2 h 37 m 39 s left | Last bid: $ 26

Antique Rose Quartz Oval Ring
Auction is live | 0 bids | 15 h 44 m 39 s left

Snowflake Obsidian Ring
Auction Starts at 1/18/2020, 8:00:00 PM

Green-striped Agate Teardrop Ring
Auction Starts at 1/19/2020, 8:00:00 PM

The `Auctions` component will iterate over the array of auctions received as a prop and display each auction in a Material-UI `ListItem` component, as shown in the following code.

`mern-marketplace/client/auction/Auctions.js`:

```
export default function Auctions(props){
    return (
     <List dense>
        {props.auctions.map((auction, i) => {
            return <span key={i}>
              <ListItem button>
                <ListItemAvatar>
                  <Avatar src={'/api/auctions/image/'+auction._id+"?"
                                          + new Date().getTime()}/>
                </ListItemAvatar>
                <ListItemText primary={auction.itemName}
                  secondary={auctionState(auction)}/>
                <ListItemSecondaryAction>
                    <Link to={"/auction/" + auction._id}>
                      <IconButton aria-label="View" color="primary">
                        <ViewIcon/>
                      </IconButton>
                    </Link>
                </ListItemSecondaryAction>
              </ListItem>
              <Divider/>
            </span>})}
        </List>
    )
}
```

For each auction item, besides displaying some basic auction details, we give the users an option to open each auction in a separate link. We also conditionally render details such as when an auction will start, whether bidding has started or ended, how much time is left, and what the latest bid is. These details of each auction's state are determined and rendered with the following code.

`mern-marketplace/client/auction/Auctions.js`:

```
const currentDate = new Date()
const auctionState = (auction)=>{
    return ( <span>
      {currentDate < new Date(auction.bidStart) &&
        `Auction Starts at ${new
Date(auction.bidStart).toLocaleString()}`}
      {currentDate > new Date(auction.bidStart) &&
        currentDate < new Date(auction.bidEnd) && <>
            {`Auction is live | ${auction.bids.length} bids |`}
            {showTimeLeft(new Date(auction.bidEnd))}
          </>}
      {currentDate > new Date(auction.bidEnd) &&
            `Auction Ended | ${auction.bids.length} bids `}
      {currentDate > new Date(auction.bidStart) &&
auction.bids.length> 0 && `
        | Last bid: $ ${auction.bids[0].bid}`}
      </span>
    )
}
```

To calculate and render the time left for auctions that have already started, we define a `showTimeLeft` method, which takes the end date as an argument and uses the `calculateTimeLeft` method to construct the time string rendered in the view. The `showTimeLeft` method is defined as follows.

`mern-marketplace/client/auction/Auctions.js`:

```
const showTimeLeft = (date) => {
    let timeLeft = calculateTimeLeft(date)
    return !timeLeft.timeEnd && <span>
      {timeLeft.days != 0 && `${timeLeft.days} d `}
      {timeLeft.hours != 0 && `${timeLeft.hours} h `}
      {timeLeft.minutes != 0 && `${timeLeft.minutes} m `}
      {timeLeft.seconds != 0 && `${timeLeft.seconds} s`} left
    </span>
}
```

This method uses the `calculateTimeLeft` method to determine the breakdown of the time left in days, hours, minutes, and seconds.

The `calculateTimeLeft` method takes the end date and compares it with the current date to calculate the difference and makes a `timeLeft` object that records the remaining days, hours, minutes, and seconds, as well as a `timeEnd` state. If the time has ended, the `timeEnd` state is set to true. The `calculateTimeLeft` method is defined as follows.

`mern-marketplace/client/auction/Auctions.js`:

```
const calculateTimeLeft = (date) => {
  const difference = date - new Date()
  let timeLeft = {}

  if (difference > 0) {
    timeLeft = {
      days: Math.floor(difference / (1000 * 60 * 60 * 24)),
      hours: Math.floor((difference / (1000 * 60 * 60)) % 24),
      minutes: Math.floor((difference / 1000 / 60) % 60),
      seconds: Math.floor((difference / 1000) % 60),
      timeEnd: false
    }
  } else {
      timeLeft = {timeEnd: true}
  }
  return timeLeft
}
```

This `Auctions` component, which renders a list of auctions with the details and a status of each, can be added to other views that will display different auction lists. If the user who's currently viewing an auction list happens to be a seller for a given auction in the list, we also want to render the option to edit or delete the auction to this user. In the next section, we will learn how to incorporate these options to edit or delete an auction from the marketplace.

Editing and deleting auctions

A seller in the marketplace will be able to manage their auctions by either editing or deleting an auction that they've created. The implementations of the edit and delete features will require building backend APIs that save changes to the database and remove an auction from the collection. These APIs will be used in frontend views to allow users to edit auction details using a form and initiate delete with a button click. In the following sections, we will learn how to add these options conditionally to the auction list and discuss the full-stack implementation to complete these edit and delete functions.

Updating the list view

We will update the code for the auctions list view to conditionally show the edit and delete options to the seller. In the `Auctions` component, which is where a list of auctions is iterated over to render each item in `ListItem`, we will add two more options in the `ListItemSecondaryAction` component, as shown in the following code.

`mern-marketplace/client/auction/Auctions.js`:

```
<ListItemSecondaryAction>
    <Link to={"/auction/" + auction._id}>
       <IconButton aria-label="View" color="primary">
          <ViewIcon/>
       </IconButton>
    </Link>
    { auth.isAuthenticated().user &&
         auth.isAuthenticated().user._id == auction.seller._id &&
      (<>
         <Link to={"/auction/edit/" + auction._id}>
            <IconButton aria-label="Edit" color="primary">
               <Edit/>
            </IconButton>
         </Link>}
        <DeleteAuction auction={auction}
onRemove={props.removeAuction}/>
      </>)
    }
</ListItemSecondaryAction>
```

The link to the edit view and the delete component are rendered conditionally if the currently signed in user's ID matches the ID of the auction seller. The implementation for the Edit view component and Delete component is similar to the `EditShop` component and `DeleteShop` component we discussed in `Chapter 7`, *Exercising MERN Skills with an Online Marketplace*. These same components will call backend APIs to complete the edit and delete actions. We will look at the required backend APIs in the next section.

Edit and delete auction APIs

To complete the edit auction and delete auction operations initiated by sellers from the frontend, we need to have the corresponding APIs in the backend. The route for these API endpoints, which will accept the update and delete requests, can be declared as follows.

`mern-marketplace/server/routes/auction.routes.js`:

```
router.route('/api/auctions/:auctionId')
  .put(authCtrl.requireSignin, auctionCtrl.isSeller,
auctionCtrl.update)
  .delete(authCtrl.requireSignin, auctionCtrl.isSeller,
auctionCtrl.remove)
router.param('auctionId', auctionCtrl.auctionByID)
```

The `:auctionId` param in the `/api/auctions/:auctionId` route URL will invoke the `auctionByID` controller method, which is similar to the `userByID` controller method. It retrieves the auction from the database and attaches it to the request object so that it can be used in the `next` method. The `auctionByID` method is defined as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

```
const auctionByID = async (req, res, next, id) => {
  try {
    let auction = await Auction.findById(id)
                              .populate('seller', '_id name')
                              .populate('bids.bidder', '_id
name').exec()
    if (!auction)
      return res.status('400').json({
        error: "Auction not found"
      })
    req.auction = auction
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve auction"
    })
  }
}
```

The auction object that's retrieved from the database will also contain the name and ID details of the seller and bidders, as we specified in the `populate()` methods. For these API endpoints, the `auction` object is used next to verify that the currently signed-in user is the seller who created this given auction by invoking the `isSeller` method, which is defined in the auction controller as follows.

`mern-marketplace/server/controllers/auction.controller.js`:

```
const isSeller = (req, res, next) => {
  const isSeller = req.auction && req.auth && req.auction.seller._id
== req.auth._id
  if(!isSeller){
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}
```

Once the seller has been verified, the `next` method is invoked to either update or delete the auction, depending on whether a PUT or DELETE request was received. The controller methods for updating and deleting auctions are similar to the previous implementations for update and delete, as we discussed for the edit shop API and delete shop API in `Chapter 7`, *Exercising MERN Skills with an Online Marketplace*.

We have the auction module for the marketplace ready with an Auction model for storing auction and bidding data and backend APIs and frontend views for creating new auctions, displaying different auction lists, and modifying an existing auction. In the next section, we will extend this module further and implement a view for individual auctions where, besides learning more about the auction, users will also be able to see live bidding updates.

Displaying the auction view

The view for displaying a single auction will contain the core functionality of the real-time auction and bidding features for the marketplace. Before getting into the implementation of real-time bidding, we will set up the full-stack slice for retrieving details of a single auction and display these details in a React component that will house the auction display, timer, and bidding capabilities. In the following sections, we will start by discussing the backend API for fetching a single auction. Then, we will look at the implementation of an Auction component, which will use this API to retrieve and display the auction details, along with the state of the auction. To give users a real-time update of the state of the auction, we will also implement a timer in this view to indicate the time left until a live auction ends.

The read auction API

To display the details of an existing auction in a view of its own, we need to add a backend API that will receive a request for the auction from the client and return its details in the response. Therefore, we will implement a read auction API in the backend that will accept a GET request with a specified auction ID and return the corresponding auction document from the `Auction` collection in the database. We will start adding this API endpoint by declaring a GET route, as shown in the following code.

`mern-marketplace/server/routes/auction.routes.js`:

```
router.route('/api/auction/:auctionId')
  .get(auctionCtrl.read)
```

The `:auctionId` param in the route URL invokes the `auctionByID` controller method when a GET request is received at this route. The `auctionByID` controller method retrieves the auction from the database and attaches it to the request object to be accessed in the `read` controller method, which is called next. The `read` controller method, which returns this auction object in response to the client, is defined as follows.

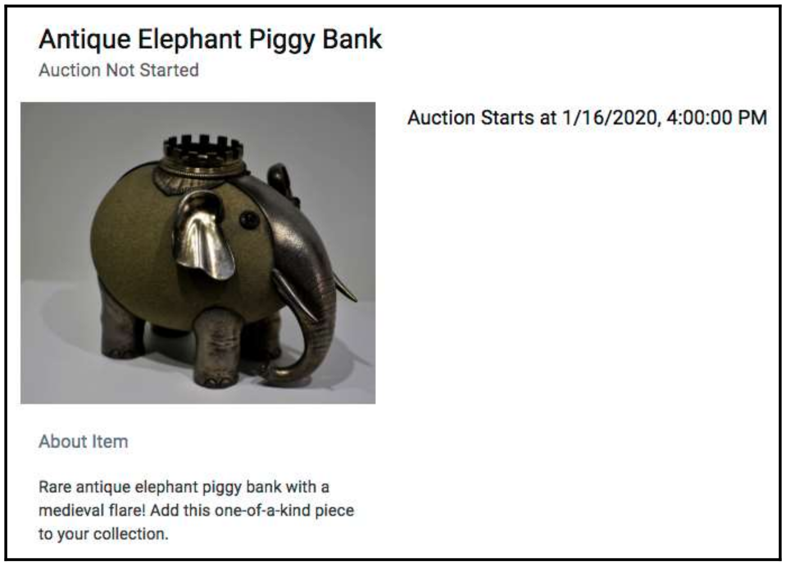
`mern-marketplace/server/controllers/auction.controller.js`:

```
const read = (req, res) => {
  req.auction.image = undefined
  return res.json(req.auction)
}
```

We are removing the image field before sending the response, since images will be retrieved as files in separate routes. With this API ready in the backend, we can now add the implementation to call it in the frontend by adding a fetch method in `api-auction.js`, similar to the other fetch methods we've discussed for completing API implementations. We will use the fetch method to call the read auction API in a React component that will render the retrieved auction details. The implementation of this React component is discussed in the next section.

The Auction component

We will implement an Auction component to fetch and display the details of a single auction to the end user. This view will also have real-time update functionalities that will render based on the current state of the auction and on whether the user viewing the page is signed in. For example, the following screenshot shows how the Auction component renders to a visitor when a given auction has not started yet. It only displays the description details of the auction and specifies when the auction will start:


Antique Elephant Piggy Bank
Auction Not Started
Auction Starts at 1/16/2020, 4:00:00 PM
About Item
Rare antique elephant piggy bank with a medieval flare! Add this one-of-a-kind piece to your collection.

The implementation of the `Auction` component will retrieve the auction details by calling the read auction API in a `useEffect` hook. This part of the component implementation is similar to the `Shop` component we discussed in `Chapter 7`, *Exercising MERN Skills with an Online Marketplace.*

The completed `Auction` component will be accessed in the browser at the `/auction/:auctionId` route, which is defined in `MainRouter` as follows.

`mern-marketplace/client/MainRouter.js`:

```
<Route path="/auction/:auctionId" component={Auction}/>
```

This route can be used in any component to link to a specific auction, as we did in the auction lists. This link will take the user to the corresponding Auction view with the auction details loaded.

In the component view, we will render the auction state by considering the current date and the given auction's bidding start and end timings. The code to generate these states, which will be shown in the view, can be added as follows.

`mern-marketplace/client/auction/Auction.js`:

```
const currentDate = new Date()
...
<span>
    {currentDate < new Date(auction.bidStart) && 'Auction Not
Started'}
    {currentDate > new Date(auction.bidStart) && currentDate < new
Date(auction.bidEnd) && 'Auction Live'}
    {currentDate > new Date(auction.bidEnd) && 'Auction Ended'}
</span>
```

In the preceding code, if the current date is before the `bidStart` date, we show a message indicating that the auction has not started yet. If the current date is between the `bidStart` and `bidEnd` dates, then the auction is live. If the current date is after the `bidEnd` date, then the auction has ended.

The `Auction` component will also conditionally render a timer and a bidding section, depending on whether the current user is signed in, and also on the state of the auction at the moment. The code to render this part of the view will be as follows.

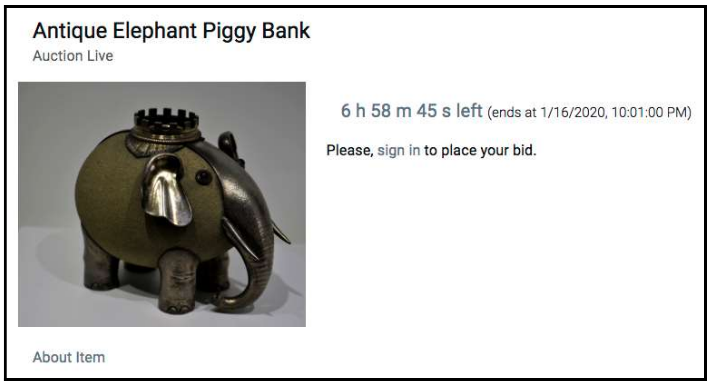
`mern-marketplace/client/auction/Auction.js`:

```
<Grid item xs={7} sm={7}>
    {currentDate > new Date(auction.bidStart)
    ? (<>
        <Timer endTime={auction.bidEnd} update={update}/>
        { auction.bids.length > 0 &&
            <Typography component="p" variant="subtitle1">
                {` Last bid: $ ${auction.bids[0].bid}`}
            </Typography>
        }
        { !auth.isAuthenticated() &&
            <Typography>
                Please, <Link to='/signin'>
                    sign in</Link> to place your  bid.
            </Typography>
        }
        { auth.isAuthenticated() &&
            <Bidding auction={auction} justEnded=
                 {justEnded} updateBids={updateBids}/>
        }
      </>)
    : <Typography component="p" variant="h6">
        {`Auction Starts at ${new
Date(auction.bidStart).toLocaleString()}`}
      </Typography>
    }
</Grid>
```

If the current date happens to be after the bid starting time, instead of showing the start time, we render the `Timer` component to show the time remaining until bidding ends. Then, we show the last bid amount, which will be the first item in the auction `bids` array if some bids were already placed. If the current user is signed in when the auction is in this state, we also render a `Bidding` component, which will allow them to bid and see the bidding history. In the next section, we will learn how to implement the Timer component we added in this view to show the remaining time for the auction.

Adding the Timer component

When the auction is live, we will give the users a real-time update of how long they have before bidding ends on this given auction. We will implement a `Timer` component and conditionally render it in the `Auction` component to achieve this feature. The timer will count down the seconds and show how much time is left to the users viewing the live auction. The following screenshot shows what the Auction component looks like when it renders a live auction to a user who is not signed in yet:


Antique Elephant Piggy Bank
Auction Live
6 h 58 m 45 s left (ends at 1/16/2020, 10:01:00 PM)
Please, sign in to place your bid.
About Item

The remaining time decreases per second as the user is viewing the live auction. We will implement this countdown feature in the `Timer` component, which is added to the `Auction` component. The `Auction` component provides it with props containing the auction end time value, as well as a function to update the auction view when the time ends, as shown in the following code.

`mern-marketplace/client/auction/Auction.js`:

```
<Timer endTime={auction.bidEnd} update={update}/>
```

The `update` function that's provided to the `Timer` component will help set the value of the `justEnded` variable from `false` to `true`. This `justEnded` value is passed to the `Bidding` component so that it can be used to disable the option to place bids when the time ends. The `justEnded` value is initialized and the `update` function is defined as follows.

`mern-marketplace/client/auction/Auction.js`:

```
const [justEnded, setJustEnded] = useState(false)
const updateBids = () => {
    setJustEnded(true)
}
```

These props will be used in the `Timer` component to calculate time left and to update the view when time is up.

In the `Timer` component definition, we will initialize the `timeLeft` variable in the state, using the end time value sent in the props from the `Auction` component, as shown in the following code.

`mern-marketplace/client/auction/Timer.js`:

```
export default function Timer (props) {
    const [timeLeft, setTimeLeft] = useState(calculateTimeLeft(new
Date(props.endTime)))
    ...
}
```

To calculate the time left until the auction ends, we utilize the `calculateTimeLeft` method we discussed previously in the *The Auctions component* section of this chapter.

To implement the time countdown functionality, we will use `setTimeout` in a `useEffect` hook in the `Timer` component, as shown in the following code.

`mern-marketplace/client/auction/Timer.js`:

```
useEffect(() => {
     let timer = null
     if(!timeLeft.timeEnd){
         timer = setTimeout(() => {
                     setTimeLeft(calculateTimeLeft(new
Date(props.endTime)))
```

```
                }, 1000)
        }else{
            props.update()
        }
        return () => {
            clearTimeout(timer)
        }
    })
```

If the time has not ended already, we will use `setTimeout` to update the `timeLeft` value after 1 second has passed. This `useEffect` hook will run after every render caused by the state update with `setTimeLeft`.

As a result, the `timeLeft` value will keep updating every second until the `timeEnd` value is `true`. When the `timeEnd` value does become `true` as the time is up, we will execute the `update` function that's sent in the props from the `Auctions` component.

To avoid a memory leak and to clean up in the `useEffect` hook, we will use `clearTimeout` to stop any pending `setTimeout` calls. To show this updating `timeLeft` value, we just need to render it in the view, as shown in the following code.

`mern-marketplace/client/auction/Timer.js`:

```
        return (<div className={props.style}>
            {!timeLeft.timeEnd ?
                <Typography component="p" variant="h6" >
                  {timeLeft.days != 0 && `${timeLeft.days} d `}
                  {timeLeft.hours != 0 && `${timeLeft.hours} h `}
                  {timeLeft.minutes != 0 && `${timeLeft.minutes} m `}
                  {timeLeft.seconds != 0 && `${timeLeft.seconds} s`} left
                  <span style={{fontSize:'0.8em'}}>
                    {`(ends at ${new
   Date(props.endTime).toLocaleString()})`}
                  </span>
                </Typography> :
                <Typography component="p" variant="h6">Auction
   ended</Typography>
            }
            </div>
        )
```

If there is time left, we render the days, hours, minutes, and seconds remaining until the auction ends using the `timeLeft` object. We also indicate the exact date and time when the auction ends. If the time is up, we just indicate that the auction ended.

In the `Auction` component we've implemented so far, we are able to fetch the auction details from the backend and render it along with the state of the auction. If an auction is in a live state, we are able to indicate the time left until it ends. When an auction is in this live state, users will also be able to place bids against the auction and see the bids being placed by other users on the platform from within this view in real-time. In the next section, we will discuss how to use Socket.IO to integrate this real-time bidding feature for all live auctions on the platform.

Implementing real-time bidding with Socket.IO

Users who are signed in to the marketplace platform will be able to take part in live auctions. They will be able to place their bids and get real-time updates in the same view while other users on the platform are countering their bids. To implement this functionality, we will integrate Socket.IO with our full-stack MERN application before implementing the frontend interface to allow users to place their bids and see the changing bidding history.

Integrating Socket.IO

Socket.IO will allow us to add the real-time bidding feature to auctions in the marketplace application. Socket.IO is a JavaScript library with a client-side module that runs in the browser and a server-side module that integrates with Node.js. Integrating these modules with our MERN-based application will enable bidirectional and real-time communication between the clients and the server.

The client-side part of Socket.IO is available as the Node module `socket.io-client`, while the server-side part is available as the Node module `socket.io`. You can learn more about Socket.IO and try their getting started tutorials at `https://socket.io`.

Before we can start using `socket.io` in our code, we will install the client and server libraries with Yarn by running the following command from the command line:

```
yarn add socket.io socket.io-client
```

With the Socket.IO libraries added to the project, we will update our backend to integrate Socket.IO with the server code. We need to initialize a new instance of `socket.io` using the same HTTP server that we are using for our application.

In our backend code, we are using Express to start the server. Therefore, we will update the code in `server.js` to get a reference to the HTTP server that our Express app is using to listen for requests from clients, as shown in the following code.

`mern-marketplace/server/server.js`:

```
import bidding from './controllers/bidding.controller'

const server = app.listen(config.port, (err) => {
  if (err) {
    console.log(err)
  }
  console.info('Server started on port %s.', config.port)
})

bidding(server)
```

Then, we will pass the reference for this server to a bidding controller function. This `bidding.controller` function will contain the Socket.IO code that's needed on the server-side to implement real-time features. The `bidding.controller` function will initialize `socket.io` and then listen on the `connection` event for incoming socket messages from clients, as shown in the following code.

`mern-marketplace/server/controllers/bidding.controller.js`:

```
export default (server) => {
    const io = require('socket.io').listen(server)
    io.on('connection', function(socket){
        socket.on('join auction room', data => {
            socket.join(data.room);
        })
        socket.on('leave auction room', data => {
            socket.leave(data.room)
        })
    })
}
```

When a new client first connects and then disconnects to the socket connection, we will subscribe and unsubscribe the client socket to a given channel. The channel will be identified by the auction ID that will be passed in the `data.room` property from the client. This way, we will have a different channel or room for each auction.

With this code, the backend is ready to receive communication from clients over sockets, and we can now add the Socket.IO integration to our frontend. In the frontend, only the auction view – specifically, the bidding section – will be using sockets for real-time communication. Therefore, we will only integrate Socket.IO in the `Bidding` component that we add to the Auction component in the frontend, as shown in the following code.

`mern-marketplace/client/auction/Auction.js`:

```
<Bidding auction={auction} justEnded={justEnded}
updateBids={updateBids}/>
```

The Bidding component takes the `auction` object, the `justEnded` value, and an `updateBids` function as props from the Auction component, and uses these in the bidding process. To start implementing the Bidding component, we will integrate sockets using the Socket.IO client-side library, as shown in the following code.

`mern-marketplace/client/auction/Bidding.js`:

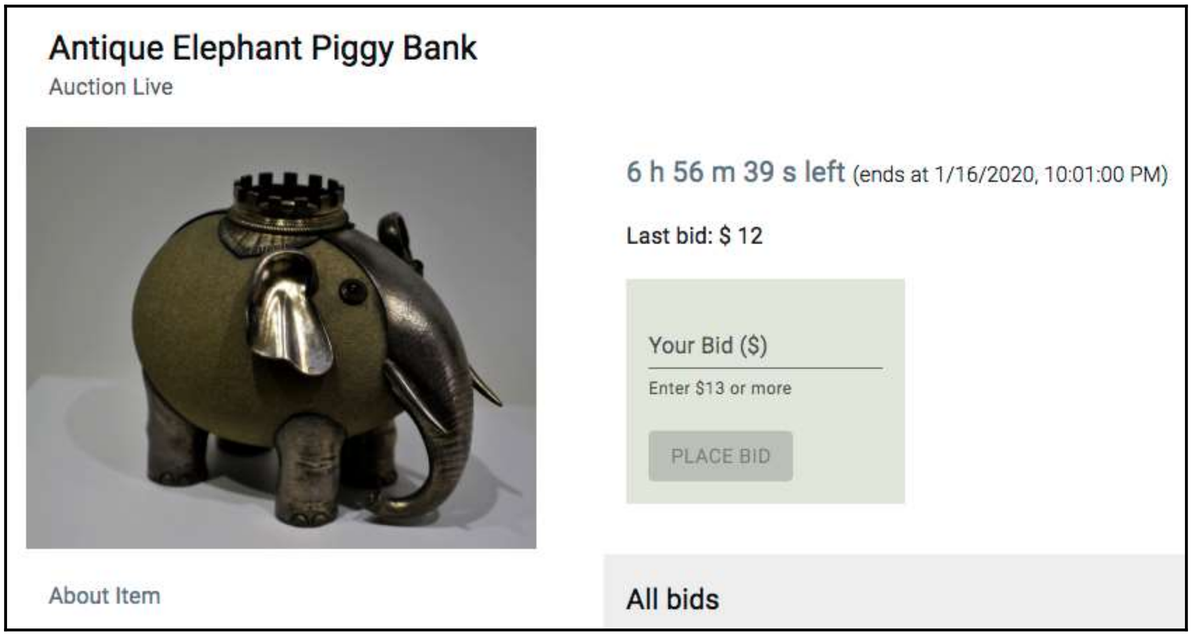
```
const io = require('socket.io-client')
const socket = io()

export default function Bidding (props) {
    useEffect(() => {
        socket.emit('join auction room', {room: props.auction._id})
        return () => {
            socket.emit('leave auction room', {
                room: props.auction._id
            })
        }
    }, [])
    ...
}
```

In the preceding code, we require the `socket.io-client` library and initialize the `socket` for this client. Then, in our `Bidding` component definition, we utilize the `useEffect` hook and the initialized `socket` to emit the *auction room joining* and *auction room leaving* socket events when the component mounts and unmounts, respectively. We pass the current auction's ID as the `data.room` value with these emitted socket events.

These events will be received by the server socket connection, resulting in subscription or unsubscription of the client to the given auction room. Now that the clients and the server are able to communicate in real-time over sockets, in the next section, we will learn how to use this capability to let users place instant bids on the auction.

Placing bids

When a user on the platform is signed in and viewing an auction that is currently live, they will see an option to place their own bid. This option will be rendered within the `Bidding` component, as shown in the following screenshot:



To allow users to place their bids, in the following sections, we will add a form that lets them enter a value more than the last bid and submit it to the server using socket communication. Then, on the server, we will handle this new bid that's been sent over the socket so that the changed auction bids can be saved in the database and the view can be updated instantly for all connected users when the server accepts this bid.

Adding a form to enter a bid

We will add the form to place a bid for an auction in the `Bidding` component that we started building in the previous section. Before we add the form elements in the view, we will initialize the `bid` value in the state, add a change handling function for the form input, and keep track of the minimum bid amount allowed, as shown in the following code.

`mern-marketplace/client/auction/Bidding.js`:

```
const [bid, setBid] = useState('')

const handleChange = event => {
        setBid(event.target.value)
}
const minBid = props.auction.bids && props.auction.bids.length> 0
                ? props.auction.bids[0].bid
                : props.auction.startingBid
```

The minimum bid amount is determined by checking the latest bid placed. If any bids were placed, the minimum bid needs to be higher than the latest bid; otherwise, it needs to be higher than the starting bid that was set by the auction seller.

The form elements for placing a bid will only render if the current date is before the auction end date. We also check if the `justEnded` value is `false` so that the form can be hidden when the time ends in real-time as the timer counts down to 0. The form elements will contain an input field, a hint at what minimum amount should be entered, and a submit button, which will remain disabled unless a valid bid amount is entered. These elements will be added to the `Bidding` component view as follows.

`mern-marketplace/client/auction/Bidding.js`:

```
{!props.justEnded && new Date() < new Date(props.auction.bidEnd) && <>
    <TextField label="Your Bid ($)"
               value={bid} onChange={handleChange}
               type="number" margin="normal"
               helperText={`Enter $${Number(minBid)+1} or
more`}/><br/>
    <Button variant="contained" color="secondary"
            disabled={bid < (minBid + 1)}
            onClick={placeBid}>Place Bid
    </Button><br/>
</>}
```

When the user clicks on the submit button, the `placeBid` function will be called. In this function, we construct a bid object containing the new bid's details, including the bid amount, bid time, and the bidder's user reference. This new bid is emitted to the server over the socket communication that's already been established for this auction room, as shown in the following code:

```
const placeBid = () => {
    const jwt = auth.isAuthenticated()
      let newBid = {
            bid: bid,
            time: new Date(),
            bidder: jwt.user
      }
      socket.emit('new bid', {
            room: props.auction._id,
            bidInfo: newBid
      })
      setBid('')
}
```

Once the message has been emitted over the socket, we will empty the input field with `setBid('')`. Then, we need to update the bidding controller in the backend to receive and handle this new bid message that's been sent from the client. In the next section, we will add the socket event handling code to complete this process to place a bid.

Receiving a bid on the server

When a new bid is placed by a user and emitted over a socket connection, it will be handled on the server so that it's stored in the corresponding auction in the database.

In the bidding controller, we will update the socket event handlers in the socket connection listener code in order to add a handler for the *new bid* socket message, as shown in the following code.

`mern-marketplace/server/controllers/bidding.controller.js`:

```
io.on('connection', function(socket){
    ...
    socket.on('new bid', data => {
        bid(data.bidInfo, data.room)
    })
})
```

In the preceding code, when the socket receives the emitted *new bid* message, we use the attached data to update the specified auction with the new bid information in a function called `bid`. The bid function is defined as follows.

`mern-marketplace/server/controllers/bidding.controller.js`:

```
const bid = async (bid, auction) => {
   try {
     let result = await Auction.findOneAndUpdate({_id:auction, $or:
[{'bids.0.bid':{$lt:bid.bid}},{bids:{$eq:[]}} ]},
                            {$push: {bids: {$each:[bid], $position:
0}}},
                            {new: true})
                            .populate('bids.bidder', '_id name')
                            .populate('seller', '_id name')
                            .exec()
     io.to(auction).emit('new bid', result)
   } catch(err) {
     console.log(err)
   }
}
```

The bid function takes the new bid details and the auction ID as arguments and performs a `findOneAndUpdate` operation on the Auction collection. To find the auction to be updated, besides querying with the auction ID, we also ensure that the new bid amount is larger than the last bid placed at position `0` of the `bids` array in this auction document. If an auction is found that matches the provided ID and also meets this condition of the last bid being smaller than the new bid, then this auction is updated by pushing the new bid into the first position of the `bids` array.

After the update to the auction in the database, we emit the *new bid* message over the `socket.io` connection to all the clients currently connected to the corresponding auction room. On the client-side, we need to capture this message in a socket event handler code and update the view with the latest bids. In the next section, we will learn how to handle and display this updated list of bids for all the clients viewing the live auction.

Displaying the changing bidding history

After a new bid is accepted on the server and stored in the database, the new array of bids will be updated in the view for all the clients currently on the auctions page. In the following sections, we will extend the `Bidding` component so that it handles the updated bids and displays the complete bidding history for the given auction.

Updating the view state with a new bid

Once the placed bid has been handled on the server, the updated auction containing the modified array of bids is sent to all the clients connected to the auction room. To handle this new data on the client-side, we need to update the `Bidding` component to add a listener for this specific socket message.

We will use an `useEffect` hook to add this socket listener to the `Bidding` component when it loads and renders. We will also remove the listener with `socket.off()` in the `useEffect` cleanup when the component unloads. This `useEffect` hook with the socket listener for receiving the new bid data will be added as follows.

`mern-marketplace/client/auction/Bidding.js`:

```
useEffect(() => {
   socket.on('new bid', payload => {
     props.updateBids(payload)
   })
   return () => {
     socket.off('new bid')
   }
})
```

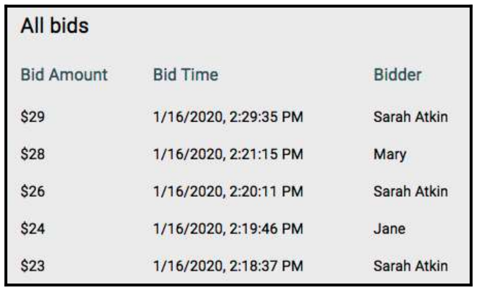
When the new auction with updated bids is received from the server in the socket event, we execute the `updateBids` function that was sent as a prop from the `Auction` component. The `updateBids` function is defined in the `Auction` component as follows:

```
const updateBids = (updatedAuction) => {
    setAuction(updatedAuction)
}
```

This will update the auction data that was set in the state of the Auction component and, as a result, rerender the complete auction view with the updated auction data. This view will also include the bidding history table, which we'll discuss in the next section.

Rendering the bidding history

In the `Bidding` component, we will render a table that displays the details of all the bids that were placed for the given auction. This will inform the user of the bids that were already placed and are being placed in real-time as they are viewing a live auction. The bidding history for an auction will render in the view as follows:



All bids

Bid Amount	Bid Time	Bidder
$29	1/16/2020, 2:29:35 PM	Sarah Atkin
$28	1/16/2020, 2:21:15 PM	Mary
$26	1/16/2020, 2:20:11 PM	Sarah Atkin
$24	1/16/2020, 2:19:46 PM	Jane
$23	1/16/2020, 2:18:37 PM	Sarah Atkin

This bidding history view will basically iterate over the `bids` array for the auction and display the bid amount, bid time, and bidder name for each bid object that's found in the array. The code for rendering this table view will be added as follows:

```
<div>
   <Typography variant="h6"> All bids </Typography>
   <Grid container spacing={4}>
       <Grid item xs={3} sm={3}>
            <Typography variant="subtitle1"
               color="primary">Bid Amount</Typography>
       </Grid>
       <Grid item xs={5} sm={5}>
            <Typography variant="subtitle1"
                color="primary">Bid Time</Typography>
       </Grid>
       <Grid item xs={4} sm={4}>
          <Typography variant="subtitle1"
             color="primary">Bidder</Typography>
       </Grid>
   </Grid>
   {props.auction.bids.map((item, index) => {
       return <Grid container spacing={4} key={index}>
                 <Grid item xs={3} sm={3}>
                    <Typography variant="body2">${item.bid}
</Typography>
                 </Grid>
                 <Grid item xs={5} sm={5}>
                    <Typography variant="body2">
                        {new Date(item.time).toLocaleString()}
                    </Typography></Grid>
                 <Grid item xs={4} sm={4}>
                    <Typography variant="body2">{item.bidder.name}
</Typography>
                 </Grid>
              </Grid>
   })}
</div>
```

We added table headers using Material-UI `Grid` components, before iterating over the `bids` array to generate the table rows with individual bid details.

When a new bid is placed by any user viewing this auction and the updated auction is received in the socket and set to state, this table containing the bidding history will update for all its viewers and show the latest bid at the top of the table. By doing this, it gives all the users in the auction room a real-time update of bidding. With that, we have a complete auction and real-time bidding feature integrated with the MERN Marketplace application.

Summary

In this chapter, we extended the MERN Marketplace application and added an auctioning feature with real-time bidding capabilities. We designed an auction model for storing auction and bidding details and implemented the full-stack CRUD functionalities that allow users to create new auctions, edit and delete auctions, and see different lists of auctions, along with individual auctions.

We added an auction view representing a single auction where users can watch and participate in the auction. In the view, we calculated and rendered the current state of the given auction, along with a countdown timer for live auctions. While implementing this timer that counts down seconds, we learned how to use `setTimeout` in a React component with the `useEffect` hook.

For each auction, we implemented real-time bidding capabilities using Socket.IO. We discussed how to integrate Socket.IO on both the client-side and the server-side of the application to establish real-time and bidirectional communication between clients and servers. With these approaches for extending the MERN stack to incorporate real-time communication functionalities, you can implement even more exciting real-time features using sockets in your own full-stack applications.

Using the experiences you've gained here building out the different features for the MERN Marketplace application, you can also grow the auctioning feature that was covered in this chapter and integrate it with the existing order management and payment processing functionalities in this application.

In the next chapter, we will expand our options with the MERN stack technologies by building an expense tracking application with data visualization features by extending the MERN skeleton.